**TRƯỜNG ĐẠI HỌC TÀI NGUYÊN VÀ MÔI TRƯỜNG HÀ NỘI**

**KHOA CÔNG NGHỆ THÔNG TIN**

****

**BÁO CÁO THỰC TẬP TỐT NGHIỆP**

**ĐỀ TÀI**

**NGHIÊN CỨU GIAO THỨC AMQP TRONG TRUYỀN TẢI DỮ LIỆU VỚI RABBITMQ**

**Đơn vị tập tốt nghiệp : Công ty cổ phần SmartOSC**

**Cán bộ hướng dẫn : Lê Văn Hiệp**

**Sinh viên thực hiện : Đặng Văn Lực**

**Lớp : ĐH6C1**

**Hệ Đại học : Chính qui**

**Khóa học : 2016 - 2020**

**Hà Nội, tháng 03/2020**

# **LỜI CẢM ƠN**

Trên thực tế không có sự thành công nào mà không gắn liền với những sự giúp đỡ mọi người dù ít hay nhiều, dù trực tiếp hay gián tiếp. Trong suốt thời gian học tập nhất là trong quá trình thực tập, em đã nhận được rất nhiều sự quan tâm giúp đỡ nhiệt tình của …

Vì thời gian, điều kiện còn có hạn, em đã cố gắng rất nhiều để hoàn thành đợt thực tập tốt nghiệp, nhưng vẫn còn nhiều hạn chế và không thể tránh khỏi những thiếu sót, mong thầy cô và các bạn có những ý kiến đóng góp để em có thể hoàn thiện và phát triển đề tài hơn.

Em xin chân thành cảm ơn!

**Sinh viên thực hiện**

**Lực**

**Đặng Văn Lực**

**MỤC LỤC**

# **DANH SÁCH CHỮ VIẾT TẮT**

|  |  |  |
| --- | --- | --- |
| **Từ viết tắt** | **Tên tiếng Anh** | **Tên tiếng Việt** |
| IaaS | Infrastructure as a Services | Hạ tầng như một dịch vụ |
| PaaS | Platform as a Services | Nền tảng như một dịch vụ |

**DANH MỤC HÌNH ẢNH**

Hình 1.1: Giao diện A…………………………………………………………………5

# **PHẦN MỞ ĐẦU**

1. **Tính cấp thiết của đề tài**

Ngày nay công nghệ thông tin phát triển rất nhanh chóng, nắm giữ vai trò quan trọng trong việc thúc đấy sự tăng trưởng kinh tế, với sự ra đời của rất nhiều công nghệ mới, các dịch vụ CNTT đáp ứng nhu cầu của người dùng cũng như là các doanh nghiệp,... Tuy nhiên, trong giai đoạn suy thoái kinh tế như hiện nay, thì việc ứng dụng một công nghệ hay một dịch vụ CNTT đáp ứng việc quản lý tốt, hiệu quả dữ liệu của riêng công ty cũng như dữ liệu khách hàng, đối tác là một trong những bài toán được ưu tiên hàng đầu cho doanh nghiệp. Để có thể quản lý được nguồn dữ liệu đó, ban đầu các doanh nghiệp phải đầu tư, tính toán rất nhiều loại chi phí như chi phí cho phần cứng, phần mềm, mạng, chi phí cho quản trị viên, chi phí bảo trì, sửa chữa,…Ngoài ra họ còn phải tính toán khả năng mở rộng, nâng cấp thiết bị; phải kiểm soát việc bảo mật dữ liệu cũng như tính sẵn sàng cao của dữ liệu. Để giải quyết vấn đề trên thì chúng ta thấy được rằng nếu có một nơi tin cậy giúp các doanh nghiệp quản lý tốt nguồn dữ liệu đó, các doanh nghiệp sẽ không còn quan tâm đến cơ sở hạ tầng, công nghệ mà chỉ tập trung chính vào công việc kinh doanh của họ thì sẽ mang lại cho họ hiệu quả và lợi nhuận ngày càng cao hơn. Trong một hệ thống phân tán (distributed system), có rất nhiều thành phần. Nếu muốn các thành phần này giao tiếp được với nhau thì chúng phải biết nhau. Nhưng điều này gây rắc rối cho việc viết code. Một thành phần phải biết quá nhiều đâm ra rất khó maintain, debug.Giải pháp ở đây là thay vì các liên kết trực tiếp, khiến các thành phần phải biết nhau thì sử dụng một liên kết trung gian qua một message broker. Với sự tham gia của message broker thì producer sẽ không hề biết consumer. Nó chỉ việc gửi message đến các queue trong message broker. Consumer chỉ việc đăng ký nhận message từ các queue này.

Tất nhiên, có thể có một giải pháp là sử dụng database để lưu các message trong các temporary table.Tuy nhiên xét về hiệu năng thì không thể bằng message broker vì một số lý do: Tần xuất trao đổi message cao sẽ làm tăng load của database, giảm performance đáng kể. Trong môi trường multithread, database cần có cơ chế lock.Lock cũng làm giảm performance. Sử dụng message broker sẽ không có vấn đề này.

1. **Tổng quan về đề tài nghiên cứu**
2. **Mục đích nghiên cứu**

* Tìm hiểu kiến trúc của giao thức AMQP
* Tìm hiểu về AMQP với RabbitMQ

1. **Đối tượng nghiên cứu và phạm vi nghiên cứu**

*Đối tượng nghiên cứu gồm có:*

* Cơ sở lý thuyết về các giao thức truyền tải dữ liệu thông dụng trong internet
* RabbitMQ và AMQP

*Phạm vi nghiên cứu:*

* Khái niệm, đặc điểm, cách thức hoạt động, mô hình triển khai, ưu điểm, nhược điểm của các giao thức truyền tải dữ liệu thông dụng trong internet.
* Khái niệm, kiến trúc hoạt động của AMQP với RabbitMQ

1. **Phương pháp nghiên cứu**

* Phương pháp nghiên cứu lý thuyết : tiến hành thu nhập các tài liệu, thông tin liên quan đến đề tài.
* Tổng hợp và phân tích tích tài liệu để đưa ra cơ sở lý thuyết về phương thức truyền tải AMQP với RabbitMQ
* Xây dựng demo về việc sử dụng giao thức AMQP với RabbitMQ

1. **Những đóng góp của báo cáo**

Đồ án hoàn thành sẽ có những đóng góp chủ yếu sau đây:

* Hệ thống hóa những vấn đề lý thuyết cơ bản về các giao thức truyền tải dữ liệu thông dụng trong internet.
* Một số vấn đề cơ bản về giao thức AMQP với RabbitMQ.
* Cách sử dụng giao thức AMQP trong RabbitMQ.

1. **Kết cấu của báo cáo**

*Đồ án gồm có 3 chương như sau:*

**Chương 1: Tổng quan về các giao thức truyền tải dữ liệu thông dụng trong internet**

Chương này trình bày tổng quan về các giao thức truyền tải dữ liệu thông dụng trong internet; ưu điểm và nhược điểm của từng phương thức truyền tải.

**Chương 2**: **AMQP với RabbitMQ.**

Chương này sẽ trình bày về microservice ,sử dụng phương thức truyền tải AMQP với RabbitMQ

**Chương 3: Xây dựng demo sử dụng phương thức AMQP trong RabbitMQ với AspCore 3.1**

Trong chương này giới thiệu về ASP.NET Core và dựng demo trên framework này**CHƯƠNG 1: TỔNG QUAN VỀ CÁC GIAO THỨC TRUYỀN TẢI DỮ LIỆU THÔNG DỤNG TRONG INTERNET**

**Internet of Things** (IoTs) là thuật ngữ rất phổ biến hiện nay, dùng để chỉ một tập hợp các thiết bị có khả năng kết nối internet và kết nối được với nhau để thực hiện một công việc nào đó. Để khai thác hết được tiềm năng của mô hình IoT, các thiết bị kết nối cần phải giao tiếp bằng các giao thức nhẹ mà không làm tiêu tốn quá nhiều tài nguyên của CPU. Dưới đây cung cấp tổng quan về 5 giao thức truyền tải dữ liệu phổ biến được sử dụng trong các mô hình Internet of Things.

* 1. **MQTT (Message Queue Telemetry Transport)**

MQTT là một giao thức mã nguồn mở để truyền các messages giữa nhiều Client (Publisher và Subscriber) thông qua một Broker trung gian, được thiết kế để đơn giản và dễ dàng triễn khai. Kiến trúc MQTT dựa trên Broker trung gian và sử dụng kết nối TCP long-lived từ các Client đến Broker.

MQTT hỗ trợ tổ chức hệ thống theo các Topics có tính phân cấp, như một hệ thống tập tin (vd: /Home/kitchen/humidity), cung cấp nhiều lựa chọn điều khiển và QoS (Quality of Service).

MQTT là một giao thức khá nhẹ nên có thể được sử dụng cho truyền thông 2 chiều thông qua các mạng có độ trễ cao và độ tin cậy thấp, nó cũng tương thích với các thiết bị tiêu thụ điện năng thấp.



Ưu điểm :

* Kết nối riêng rẽ
* Khả năng mở rộng



Nhược điểm :

* Thời gian tách biệt (Time decoupling)
* Đồng bộ riêng rẽ (Synchronization decoupling)
* Điểm trung gian (broker) không cần thông báo về trạng thái gửi thông điệp. Do đó không có cách nào để phát hiện xem thông điệp đã gửi đúng hay chưa.
* Publisher không hề biết gì về trạng thái của subscribe và ngược lại. Vậy làm sao chúng ta có thể đảm bảo mọi thứ đều ổn
* Những kẻ xấu (malicious publisher) có thể gửi những thông điệp xấu, và các subcribers sẽ truy cập vào những thứ mà họ không nên nhận

Cơ chế tổng quan

* MQTT hoạt động theo cơ chế client/server, nơi mà mỗi cảm biến là một khách hàng (client) và kết nối đến một máy chủ, có thể hiểu như một điểm trung gian (broker), thông qua giao thức TCP (Transmission Control Protocol). Broker chịu trách nhiệm điều phối tất cả các thông điệp giữa phía gửi đến đúng phía nhận.
* MQTT là giao thức định hướng bản tin. Mỗi bản tin là một đoạn rời rạc của tín hiệu và broker không thể nhìn thấy. Mỗi bản tin được publish một địa chỉ, có thể hiểu như một kênh (Topic). Client đăng kí vào một vài kênh để nhận/gửi dữ liệu, gọi là subscribe. Client có thể subcribe vào nhiều kênh. Mỗi client sẽ nhận được dữ liệu khi bất kì trạm nào khác gửi dữ liệu vào kênh đã đăng kí. Khi một client gửi một bản tin đến một kênh nào đó gọi là publish

Ví dụ :



* 1. **CoAP (Constrained Applications Protocol)**

CoAP là một giao thức truyền tải tài liệu theo mô hình client/server dự trên internet tương tự như giao thức HTTP nhưng được thiết kế cho các thiết bị ràng buộc. Giao thức này hỗ trợ một giao thức one-to-one để chuyển đổi trạng thái thông tin giữa client và server.

CoAP sử dụng UDP (User Datagram Protocol), không hỗ trợ TCP, ngoài ra còn hỗ trợ địa chỉ broadcast và multicast, truyền thông CoAP thông qua các datagram phi kết nối (connectionless) có thể được sử dụng trên các giao thức truyền thông dựa trên các gói.

UDP có thể dễ dàng triển khai trên các vi điều khiển hơn TCP nhưng các công cụ bảo mật như SSL/TSL không có sẵn, tuy nhiên ta có thể sử dụng Datagram Transport Layer Security (DTLS) để thay thế..



Cơ chế tổng quan :

* CoAP theo mô hình client/server. Client gửi yêu cầu đến máy chủ, sau đó máy chủ gửi lại phản hồi. Client có thể GET, PUT, POST và DELETE các tài nguyên.
* CoAP có tính linh động và hỗ trợ đàm phán nội dung. Điều này cho phép client và máy chủ có thể nâng cấp, thêm mới một cách độc lập mà ko ảnh hưởng gì đến phía còn lại.
* CoAP đưa ra các yêu cầu quan sát tài nguyên. Cả hai bên đều có thể tác động hoặc xóa các yêu cầu quan sát. Khi cờ quan sát được thiết lập, máy chủ vẫn có thể tiếp tục hồi đáp sau khi các dữ liệu đã truyền đi.
* Với CoAP, máy chủ cung cấp một hệ thống các tài nguyên cho phép client khám phá tài nguyên và các loại phương tiện truyền thông.
* Trong CoAP một nút cảm biến thường là một máy chủ. Chúng có khả năng nhận các gói tin gửi đến để hoạt động đúng đằng sau NAT, thiết bị đầu tiên phải gửi yêu cầu đến máy chủ, như được thực hiện trong LWM2M, cho phép các router liên kết chúng lại.

Hạn chế :

* Sử dụng CoAP khi HTTP quá tải băng thông : Việc sử dụng CoAP không lớn như HTTP, vì vậy nó có thể giới hạn các tùy chọn phần mềm và phần cứng của ta. Có các giải pháp để chuyển đổi các thông báo CoAP sang và từ HTTP giúp các giải pháp CoAP tương thích hơn.
  1. **AMQP (Advanced Message Queue Protocol)**

****

AMQP là một giao thức làm trung gian cho các gói tin trên lớp ứng dụng với mục đích thay thế các hệ thống truyền tin độc quyền và không tương thích. Các tính năng chính của AMQP là định hướng message, hàng đợi, định tuyến (bao gồm point-to-point và publish-subscribe) có độ tin cậy và bảo mật cao. Các hoạt động sẽ được thực hiện thông qua broker, nó cung cấp khả năng điều khiển luồng (Flow Control).

Một trong các Message Broker phổ biến là RabbitMQ, được lập trình bằng ngôn ngữ Erlang, RabbitMQ cung cấp cho lập trình viên một phương tiện trung gian để giao tiếp giữa nhiều thành phần trong một hệ thống lớn.

Không giống như các giao thức khác, AMQP là một giao thức có dây (wire-protocol), có khả năng diễn tả các message phù hợp với định dạng dữ liệu, có thể triển khai với rất nhiều loại ngôn ngữ lập trình.

AMQP kết nối qua:

* Tổ chức – ứng dụng trong các tổ chức khác nhau
* Công nghệ – ứng dụng trên các nền tảng khác nhau
* Thời gian: Hỗ trợ giao tiếp bất đồng bộ
* Không gian – hoạt động ở khoảng cách xa hoặc trên các mạng nghèo nàn



* 1. **DDS (Data Distribution Service)**



DDS là một ngôn ngữ trung gian dựa vào dữ liệu tập trung được sử dụng để cho phép khả năng mở rộng, thời gian thực, độ tin cậy cao và trao đổi dữ liệu tương tác.

Đây là một giao thức phi tập trung (broker-less) với truyền thông ngang hàng trực tiếp theo kiểu peer-to-peer giữa các publishers và subscribers và được thiết kế để trở thành một ngôn ngữ và hệ điều hành độc lập. DDS gửi và nhận dữ liệu, sự kiện, và thông tin lệnh trên UDP nhưng cũng có thể chạy trên các giao thức truyền tải khác như IP Multicast, TCP / IP, bộ nhớ chia sẻ … DDS hỗ trợ các kết nối được quản lý many-to-many theo thời gian thực và ngoài ra còn hỗ trợ dò tìm tự động (automatic discovery). Các ứng dụng sử dụng DDS cho truyền thông được tách riêng và không yêu cầu sự can thiệp từ các ứng dụng của người dùng, có thể đơn giản hóa việc lập trình mạng phức tạp. Các tham số QoS được sử dụng để xác định các cơ chế tự dò tìm của nó được thiết lập một lần.

* 1. **XMPP (Extensible Messaging và Presence Protocol)**



XMPP (trước đây gọi là “Jabber”) là giao thức truyền thông dùng cho định hướng tin nhắn trung gian dựa trên ngôn ngữ XML.

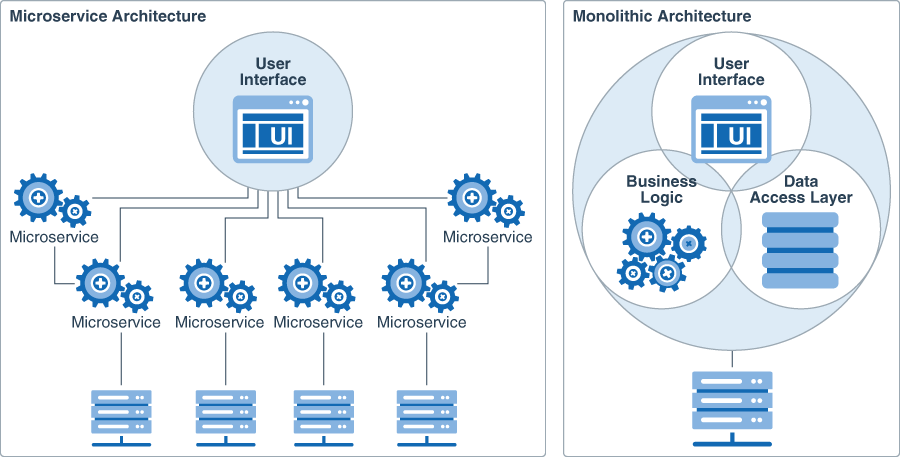
XMPP là mô hình phân quyền client-server phi tập trung, được sử dụng cho các ứng dụng nhắn tin văn bản. Có thể nói XMPP gần như là thời gian thực và có thể mở rộng đến hàng trăm hàng nghìn nút. Dữ liệu nhị phân phải được mã hóa base64 trước khi nó được truyền đi trong băng tần. XMPP tương tự như MQTT, có thể chạy trên nền tảng TCP.

# **CHƯƠNG 2: AMQP VỚI RabbitMQ**

**2.1 Microservice là gì**

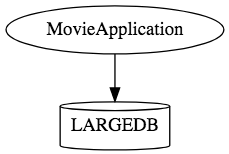
Thực tế có nhiều định nghĩa khác nhau đối với Microservices nhưng hiểu theo cách đơn giản thì, microservice là một kiếu kiến trúc phần mềm. Các module trong phần mềm này được chia thành các service rất nhỏ (microservice). Mỗi service sẽ được đặt trên một server riêng -> dễ dàng để nâng cấp và scale ứng dụng.

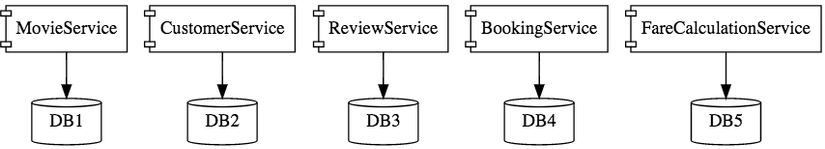
### *2.1.1. Kiến trúc*



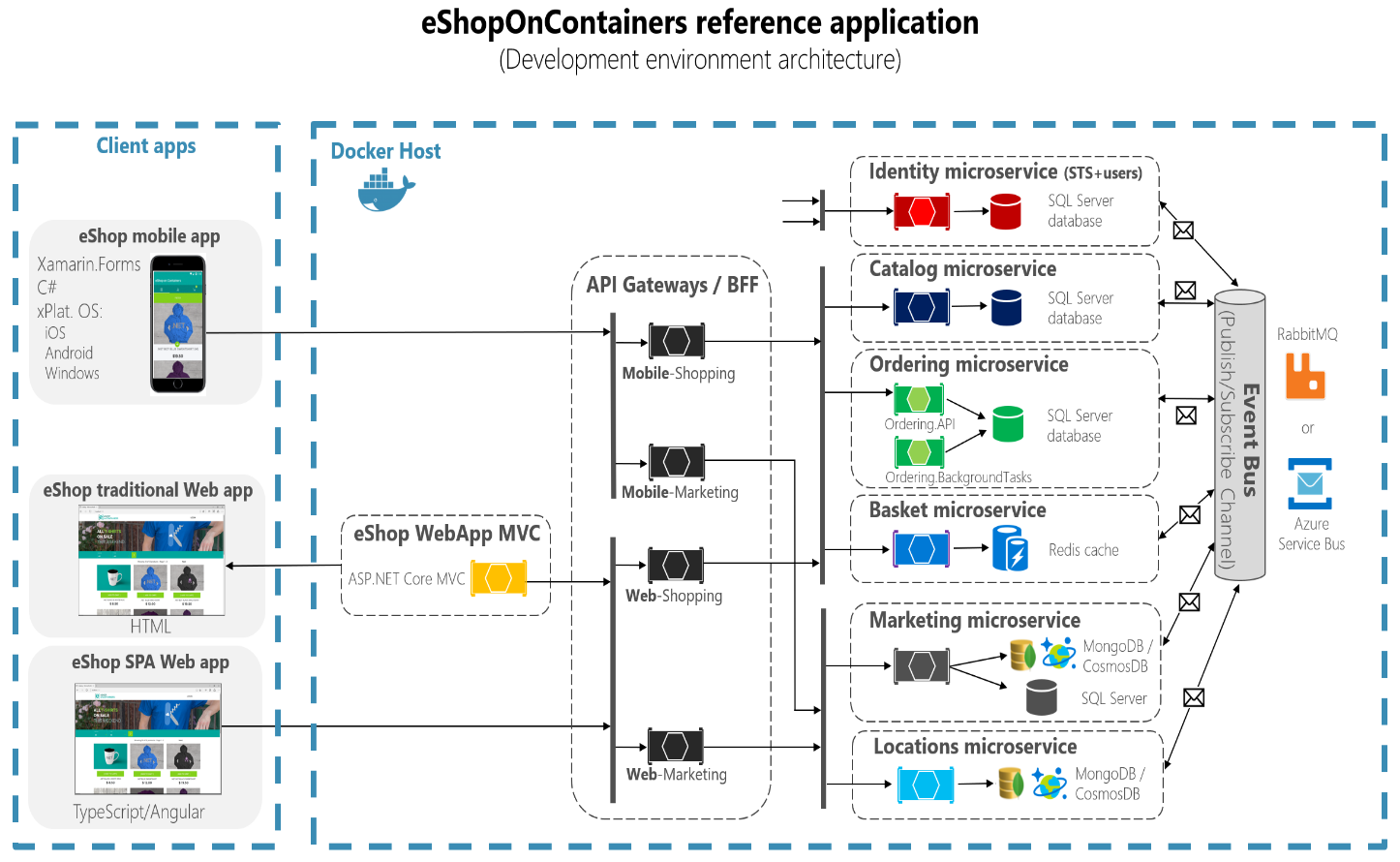
Khác biệt với kiến trúc Monolith (một khối), hay vì gom tất cả module thành một khối (monolith), ta tách các module thành những service siêu nhỏ. Mỗi service sẽ được đặt trên một server riêng (Có thể dùng server cloud như AWS hoặc Azure), giao tiếp với nhau thông qua mạng (Gửi nhận message qua giao thức HTTP hoặc sử dụng MessageQueue)...

Hình dưới đây sẽ minh họa cho việc phần mềm được xây dựng theo kiến trúc Monolith, một ứng dụng sẽ chứa tất cả các thành phần



Còn hình dưới sẽ minh họa việc ứng dụng ở trên khi được xây dựng theo kiến trúc Microservices

Kiến trúc Microservice bao gồm một số thành phần nhỏ, được thiết kế tốt và tương tác qua các message.



### *2.1.2. Ưu điểm của Microservices*

+ Giảm thiểu sự gia tăng phức tạp rối rắm hệ thống lớn.

+ Chia nhỏ ứng dụng một khối cồng kềnh thành các dịch vụ nhỏ dễ quản lý, bảo trì nâng cấp, tự do chọn, nâng cấp công nghệ mới.

+ Mỗi dịch vụ nhỏ sẽ định ra ranh giới rõ ràng dưới dạng RPC hay API hướng thông điệp.

+ Microservice thúc đẩy tách rạch ròi các khối chức năng (loose coupling - high cohesion), điều rất khó thực hiện với ứng dụng một khối. Nếu muốn loose coupling - high cohesion trong ứng dụng một khối, sẽ phải thiết kế theo Design Pattern (Gang Of Four) và liên tục tái cấu trúc (refactor)

+ Mỗi dịch vụ nhỏ sẽ phát triển dễ hơn, nhanh hơn, dễ viết mã kiểm thử tự động.

### *2.1.3. Nhược điểm của Microservices*

+ Microservice nhấn mạnh kích thước nhỏ gọn của dịch vụ. Một số lập trình đề xuất dịch vụ siêu nhỏ cỡ dưới 100 dòng code. Chia quá nhiều sẽ dẫn đến manh mún, vụn vặt, khó kiểm soát. Việc lưu dữ liệu cục bộ bên trong những dịch vụ quá nhỏ sẽ khiến dữ liệu phân tán quá mức cần thiết.

+ Phải xử lý sự cố khi kết nối chậm, lỗi khi thông điệp không gửi được hoặc thông điệp gửi đến nhiều đích đến vào các thời điểm khác nhau.

+ Đảm bảo giao dịch phân tán (distributed transaction) cập nhật dữ liệu đúng đắn (all or none) vào nhiều dịch vụ nhỏ khác nhau khó hơn rất nhiều, đôi khi là không thể so với đảm bảo giao dịch cập nhật vào nhiều bảng trong một cơ sở dữ liệu trung tâm.

+ Theo nguyên tắc CAP (CAP theorem) thì giao dịch phân tán sẽ không thể thỏa mãn cả 3 điều kiện: consistency (dữ liệu ở điểm khác nhau trong mạng phải giống nhau), availablity (yêu cầu gửi đi phải có phúc đáp), partition tolerance (hệ thống vẫn hoạt động được ngay cả khi mạng bị lỗi). Những công nghệ cơ sở dữ liệu phi quan hệ (NoSQL) hay môi giới thông điệp (message broker) tốt nhất hiện nay cũng chưa vượt qua nguyên tắc CAP

+ Kiểm thử tự động một dịch vụ trong kiến trúc microservices đôi khi yêu cầu phải chạy cả các dịch vụ nhỏ khác mà nó phụ thuộc. Do đó khi phân rã ứng dụng một khối thành microservices cần luôn kiểm tra mức độ ràng buộc giữa các dịch vụ mềm dẻo hơn hay cứng nhắc - lệ thuộc hơn. Nếu ràng buộc ít đi, lỏng leo hơn, bạn đi đúng hướng và ngược lại

+ Nếu các dịch vụ nhỏ thiết kế phục thuộc vào nhau theo chuỗi. A gọi B, B gọi C, C gọi D. Nếu một mắt xích có giao tiếp API thay đổi, liệu các mắt xích khác có phải thay đổi theo , việc bảo trì, kiểm thử sẽ phức tạp tương tự ứng dụng một khối. Chính vì thế việc thiết kế dịch vụ không tốt sẽ ảnh hưởng lan truyền đến các dịch vụ khác.

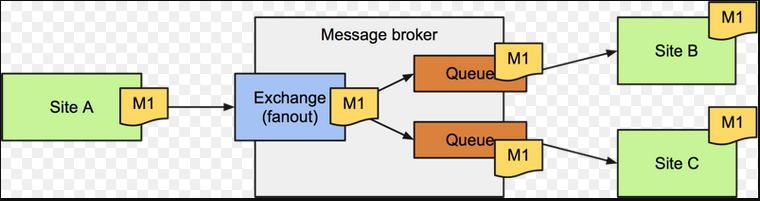
* 1. **RabbitMQ là gì ? Tại sao sử dụng RabbitMQ**

RabbitMQ là một message-queuing software có thể được biết đến như là một người vận chuyển message trung gian hoặc một người quản lí các queue. Nói một cách đơn giản, nó là một phần mềm nơi các queue được định nghĩa, phục vụ cho ứng dụng với mục đích vận chuyển một hoặc nhiều message

Vậy, tại sao chúng ta phải cần đến RabbitMQ? Hãy tưởng tượng, hiện bạn đang có một web service, phải nhận rất rất nhiều request mỗi giây, mà lại phải đảm bảo rằng không có bất cứ một request nào bị mất. Và web service của bạn luôn luôn sẵn sàng tiếp nhận request mới thay vì locked bởi đang xử lí request trước đó. Vậy ý tưởng ở đây là đặt chúng vào một queue giữa web service và processing service. Lúc này sẽ đảm bảo rằng 2 process sẽ hoàn toàn tách rời nhau. Ngoài ra, queue sẽ lưu trữ những request, không bị thiếu sót request nào khi số lượng của chúng trở nên vô cùng lớn

* 1. **Kiến trúc của RabbitMQ , các khái niệm cần biết trong RabbitMQ**

### *2.3.1. Kiến trúc của RabbitMQ*

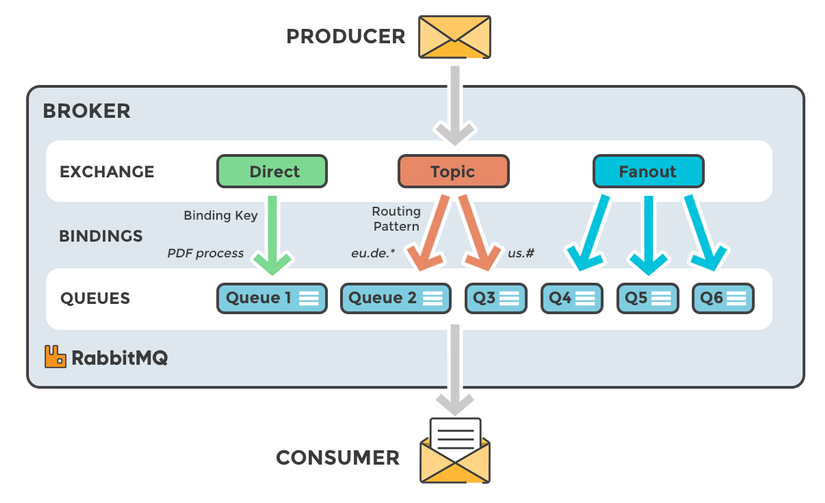


Có thể hiểu message broker gần như bưu điện. Site A theo cách gọi của rabbitmq là producer (người gửi thông điệp). Site B và Site C theo cách gọi của rabbitmq là consumer (người nhận thông điệp). Producer connect đến message broker để đẩy message. Message sẽ đi qua message broker để đến được consumer. Cấu trúc của message broker chỉ gồm hai phần exchange và queue.

Exchange có nhiều loại. Trong hình vẽ trên exchange type là fanout. Lựa chọn các exchange type khác nhau sẽ dẫn đến khác đối xử khác nhau của message broker với thông điệp nhận được từ producer. Exchange được bind (liên kêt) đến một số queue nhất định. Với exchange type là fanout, message sẽ được broadcast đến các queue được bind với exchange. Consumer sẽ connect đến message broker để lấy message từ các queue.

### *2.3.1. Những Khái Niệm Cơ Bản*

|  |  |
| --- | --- |
| **Producer** | Ứng dụng gửi message |
| **Consumer** | Ứng dụng nhận message |
| **Queue** | Lưu trữ messages |
| **Message** | Thông tin truyền từ Producer đến Consumer qua RabbitMQ |
| **Connection** | Một kết nối TCP giữa ứng dụng và RabbitMQ broker |
| **Channel** | Là nơi nhận message được publish từ Producer và đẩy chúng vào queue dựa vào quy tắc của từng loại Exchange. Để nhận được message, queue phải được nằm trong ít nhất 1 Exchange |
| **Exchange** | Là nơi nhận message được publish từ Producer và đẩy chúng vào queue dựa vào quy tắc của từng loại Exchange. Để nhận được message, queue phải được nằm trong ít nhất 1 Exchange |
| **Binding** | Đảm nhận nhiệm vụ liên kết giữa Exchange và Queue |
| **Routing key** | Một key mà Exchange dựa vào đó để quyết định cách để định tuyến message đến queue. Có thể hiểu nôm na, Routing key là địa chỉ dành cho message |
| **AMQP** | Giao thức Advance Message Queuing Protocol, là giao thức truyền message trong RabbitMQ |
| **User** | Để có thể truy cập vào RabbitMQ, chúng ta phải có username và password. Trong RabbitMQ, mỗi user được chỉ định với một quyền hạn nào đó. User có thể được phân quyền đặc biệt cho một Vhost nào đó |
| **Virtual host/Vhost** | Cung cấp những cách riêng biệt để các ứng dụng dùng chung một RabbitMQ instance. Những user khác nhau có thể có các quyền khác nhau đối với vhost khác nhau. Queue và Exchange có thể được tạo, vì vậy chúng chỉ tồn tại trong một vhost |



Hình 1. Sơ đồ vận chuyển message trong RabbitMQ

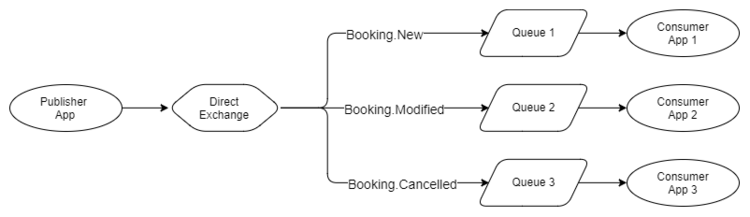
### *2.3.1. Các loại định tuyến cơ bản của RabbitMQ*

#### 2.3.1.1. Default Exchange và Direct Exchange

**Direct Exchanges** định tuyến thông điệp bằng cách sử dụng Routing Key của thông điệp. Routing key được thiết đặt bởi các publisher tương ứng với thông điệp. Chúng là các chuỗi gồm nhiều từ được phân cách với nhau bằng dấu chấm. Ví dụ như, *"booking.new"*, *"booking.modified"* và *"booking.cancelled"*.

Liên kết giữa một Queue hoặc một Exchange với một **Direct Exchange** chứa một binding key, giá trị này sẽ được so sánh chính xác tuyệt đối.

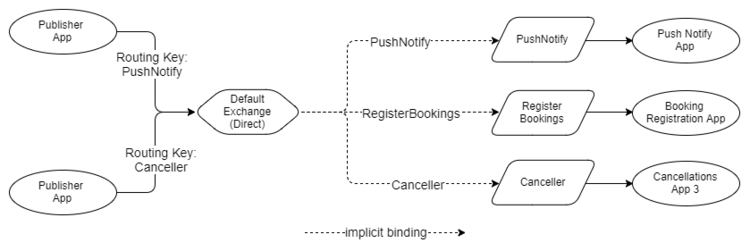
*Hình 2: Direct exchanges định tuyến bằng cách so sánh chính xác Routing Key và Binding Key*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_002.png)

**Direct exchange** là loại exchange nhanh thứ hai vì chúng chỉ thực hiện so sánh chính xác các chuỗi với nhau.

Có một loại exchange đặc biệt được gọi là Default exchange, nó cũng là một loại Direct echange. **Default exchange** có mối liên kết ngầm với tất cả các hàng đợi trong *virtual host* của nó. Mối liên kết ngầm tới mỗi queue sẽ có binding key là tên của các queue đó. Điều này nghĩa là bạn có thể gửi thông điệp trực tiếp đến các queue bằng tên của queue tương ứng.

*Hình 3: Default exchange có mối liên kết ngầm đến mỗi queue*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_003.png)

Điều này có thể hữu ích nếu publisher muốn chọn chính xác consumer mà nó muốn xử lý thông điệp của nó, thay vì dựa vào các liên kết được cấu hình bởi các consumer. Thông thường, chúng ta sẽ muốn tách rời hoàn toàn publisher và subscriber với nhau, đây là khả năng mà các loại exchange khác đều có thể cung cấp. Nhưng trong trường hợp bạn muốn gửi thông điệp theo kiểu point-to-point thì Default exchange sẽ cung cấp cho bạn khả năng này.

#### 2.3.1.3. Topic Exchange

Các exchange này cũng định tuyến bằng routing key, nhưng các **Topic Exchange** cung cấp việc sử dụng hai loại ký tự đại diện trong Khóa Binding.

Ký tự \* đại diện khớp với một từ trong routing key. Ví dụ: routing key "booking.new" có hai từ. Ký tự # phù hợp với bất kỳ số lượng từ.

Ví dụ: giả sử chúng ta có các routing key sau:

* booking.new
* booking.modified
* booking.cancelled
* extras.car.new
* extras.car.modified
* extras.car.removed
* extras.hotel.new
* extras.hotel.modified
* extras.hotel.removed

Chúng ta có thể tạo các liên kết với các khóa liên kết sau:

* booking.new - so sánh chính xác từ khóa
* extras.\*.modified - tất cả các sửa đổi đối với tính năng bổ sung khi đặt phòng (xe hơi hoặc khách sạn)
* extras.# - tất cả các tính năng bổ sung
* #.new - tất cả các đặt phòng mới và tính năng bổ sung

Với thiết kế cẩn thận các routing key và khóa liên kết, chúng ta có thể thêm các routing key mới mà không cần cập nhật các ràng buộc hiện có, làm cho hệ thống trở nên mạnh mẽ khi đối mặt với sự thay đổi.

**Topic Exchange** cho phép bạn định cấu hình một exchange duy nhất cho một ứng dụng để gửi thông điệp đến, sử dụng định tuyến để đảm bảo rằng các thông điệp đến đúng consumer. Điều này giúp đơn giản hóa cấu hình và triển khai các ứng dụng xuất bản và tiêu thụ.

Lưu ý rằng **Topic Exchange** chậm lại khi số lượng ràng buộc tăng lên.

#### 2.3.1.4. Headers Exchange

Đây là exchange có tính năng mạnh mẽ nhất, nhưng cũng chậm nhất trong các loại exchange. Thực tế này cần phải được tính đến vì bạn có thể có vấn đề với mở rộng hệ thống với loại exchange này. **Header exchane** bỏ qua routing key và thay vào đó phân tích các tiêu đề của thông điệp. Mỗi liên kết với một **Header Exchange** có thể bao gồm nhiều đối sánh tiêu đề mà ANY hoặc ALL bắt buộc phải khớp.

Giả sử ứng dụng của bạn publish lên một exchange một tập hợp các thông điệp khác nhau tạo thành nhật ký thay đổi thời gian thực cho phép tích hợp với các hệ thống khác. Mỗi thông điệp có các tiêu đề thư sau:

* entity.type (đặt chỗ, hành khách, hành lý, thú cưng)
* change.type (mới, sửa đổi, hủy bỏ, xóa, di chuyển)
* agent.id
* client.id

Chúng ta có thể tạo các liên kết sau:

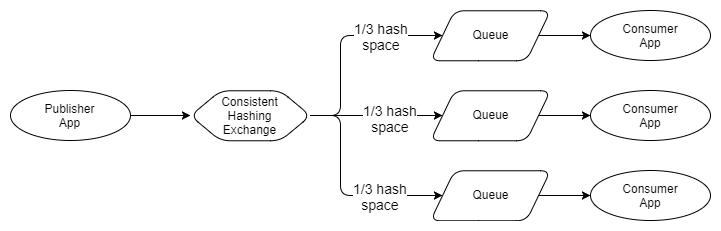
* entity.type=booking, change.type=cancelled, x-match=all. Tôi muốn tất cả các thông điệp đặt phòng bị hủy.
* entity.type=passenger, x-match=all. Tôi muốn tất cả các thông điệp hành khách.
* entity.type=pet, change.type=new, x-match=all. Tôi muốn tất cả các thông điệp thú cưng mới được thêm vào.
* agent.id=2, client.id=1001, x-match=any. Tôi muốn tất cả các thông điệp liên quan đến đại lý du lịch cụ thể hoặc khách hàng cuối.

Như bạn có thể thấy, **Header Exchange** khá mạnh mẽ.

Consistent Hashing Exchange

**Consistent Hashing Exchange** cho phép chúng ta phân vùng một hàng đợi thành nhiều hàng đợi và phân phối thông điệp giữa chúng thông qua việc băm routing key, tiêu đề thư hoặc thuộc tính thông điệp.

Hình 4:

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_004.png)

Điều này cung cấp cho chúng ta các mẫu như bảo đảm xử lý theo thứ tự và dữ liệu cục bộ, hai pattern bạn sẽ thấy bên dưới trong phần các pattern.

Có một số vấn đề với **Consistent Hashing Exchange** mặc dù. Đầu tiên, RabbitMQ không giúp bạn điều phối khách hàng của mình qua các hàng đợi được phân vùng như Kafka. Vì vậy, đó là xuống để bạn quản lý bằng cách nào đó. Kafka cung cấp cho bạn điều này ra khỏi hộp.

Các vấn đề tiềm năng khác là:

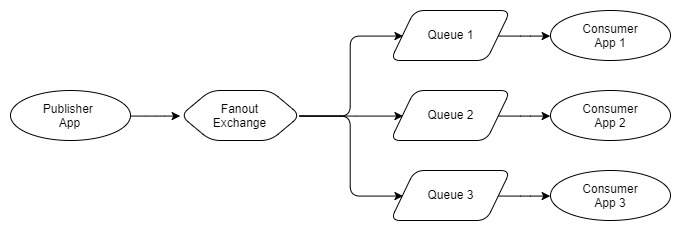
* Thứ bạn băm (routing key, message header hoặc các thuộc tính) không có qui chuẩn nào để tạo phân phối đồng đều. Nếu bạn chỉ có bốn giá trị khác nhau thì bạn có thể gặp xui xẻo và tất cả đi đến một hàng đợi.
* Nếu bạn có tương đối ít hàng đợi thì phân phối có thể không đồng đều trở lại.

Các hệ thống phân tán khác giải quyết điều này bằng cách sử dụng khái niệm các nút ảo. Ví dụ, để ngăn phân phối mất cân bằng, Rịa và Cassandra có các nút ảo có số lượng lớn hơn nhiều so với các nút vật lý và chúng phân phối các nút ảo này qua các nút vật lý. Bằng cách đó, chúng có được phân phối tốt hơn khi một cụm có tương đối ít nút vật lý. RabbitMQ không có khái niệm này vì vậy hãy chú ý đến việc phân phối thông điệp.

#### 2.3.1.4. Fanout Exchanges

Những exchange này cung cấp cấu trúc liên kết đăng ký xuất bản điển hình. Một thông điệp được gửi đến một Fanout Exchange sẽ được phát tới cho tất cả các hàng đợi và các exchange có liên kết đến exchange này.

*Hình 1:*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_001.png)

Trong sơ đồ trên, mỗi consumer độc lập với những consumer khác và nhận được các bản sao riêng của tất cả các thông điệp. Để mở rộng ứng dụng Consumer App 1, nhiều phiên bản ứng dụng đó sẽ cần được triển khai, tiêu thụ từ cùng một Queue 1.

**Fanout Exchange** là exchange nhanh nhất trong các exchange vì chúng không cần phải kiểm tra bất kỳ routing key nào hoặc kiểm tra header của thông điệp. Mặc dù một exchange có thể gửi một thông điệp duy nhất cho nhiều hàng đợi, trong thực tế nó không thường xuyên sao chép các thông tin. Nó có thể lưu thông điệp vào cơ sở dữ liệu **Mnesia** và chỉ cần đăng ký một con trỏ tới thông điệp trong mỗi hàng đợi.

#### 2.3.1.4. Dead Letter Exchange

Chúng ta có thể định cấu hình hàng đợi để đẩy một thông điệp và gửi nó đến một exchange được cấu hình theo một trong ba điều kiện:

* Hàng đợi đã đạt đến giới hạn số lượng thông điệp. Thông báo ở đầu hàng đợi (thông điệp cũ nhất) được đẩy ra và gửi đến **Dead Letter Exchange** đã được cấu hình (DLX). Vì vậy, khi một thông điệp mới đến một hàng đợi đầy đủ, về cơ bản nó sẽ loại bỏ thông điệp cũ nhất và được thêm vào hàng đợi một cách an toàn.
* Hàng đợi đã đạt đến giới hạn kích thước (byte). Một lần nữa, thông điệp cũ nhất bị đẩy ra.
* Hàng đợi đã được cấu hình với một giới hạn tồn tại của thông điệp (TTL) và một thông điệp đã đạt đến giới hạn đó. Một thông điệp đã được cấu hình với TTL của chính nó và nó đã đạt đến khoảng thời gian đó trong hàng đợi.

Thông điệp chỉ bị coi là đã chết khi đứng đầu hàng đợi. Vì vậy, các thông điệp đã vượt qua TTL của chúng chỉ được chuyển tiếp đến DLX khi chúng đến đầu hàng đợi. Điều này rất quan trọng!

DLX chỉ là một exchange thông thường, bạn có thể tạo một trong bốn loại và liên kết bất kỳ hàng đợi hoặc exchange khác với nó.

Chức năng thư chết của RabbitMQ không chỉ cung cấp một lối thoát cho các thông điệp trong tình trạng bị mất. Nó có thể được sử dụng để retry và delay hàng đợi như chúng ta sẽ thấy trong phần ví dụ một số pattern.

#### 2.3.1.4. Ephemeral Exchanges và Queue

Exchange có thể được cấu hình để tự động xóa tất cả các liên kết hàng đợi đã được gỡ bỏ. Các liên kết hàng đợi có thể được loại bỏ bằng cách chỉ cần loại bỏ các liên kết hoặc loại bỏ hàng đợi.

Hàng đợi có thể được cấu hình để tự động xóa sau khi tất cả consumer đã ngừng sử dụng hàng đợi. Điều này có thể là do consumer đã hủy đăng ký hoặc kênh đã đóng.

Hàng đợi có thể được cấu hình thành hàng đợi độc quyền. Điều này có nghĩa là chỉ consumer đã khai báo hàng đợi mới có thể tiêu thụ nó và một khi consumer hủy hoặc đóng kênh, hàng đợi sẽ tự động xóa.

Hàng đợi có thể được cấu hình với một hàng đợi TTL. Khi hàng đợi không được sử dụng trong khoảng thời gian TTL, nó sẽ bị xóa. Không sử dụng có nghĩa là không có consumer hoạt động đăng ký.

Ephemeral Exchanges và Queue có thể được sử dụng cho các mẫu như delay queue, retry queue và reply-to queue, như chúng ta sẽ thấy trong phần ví dụ về các pattern.

#### 2.3.1.4. Alternate Exchange - Exchange thay thế

Mỗi exchange có thể được cấu hình với một Alternate Exchange. Khi một exchange không thể định tuyến một thông điệp vì không có liên kết hoặc không có liên kết nào khớp với thông điệp, thì exchange đó sẽ định tuyến thông điệp đến Alternate Exchange của nó.

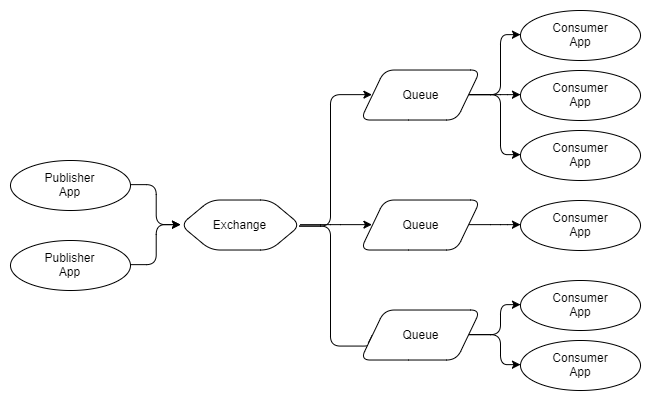
Điều này cung cấp cho chúng ta một cách không làm mất các thông điệp có thể bị mất do routing key xấu hoặc cấu trúc liên kết định tuyến xấu. Nhưng nó cũng cho phép tạo ra các pattern định tuyến mới mà bốn loại echange không cung cấp. Chúng ta sẽ thấy trong ví dụ về các pattern cách mà Alternate Exchange có thể được sử dụng cho các tình huống định tuyến khác nhau.

### 2.3.2. Một số messaging pattern khi sử dụng RabbitMQ AMQP

#### 2.3.2.1. Alternate Exchange - Exchange thay thế

Sử dụng Fanout Exchange để truyền tất cả thông điệp cho tất cả consumer.

*Hình 5:*

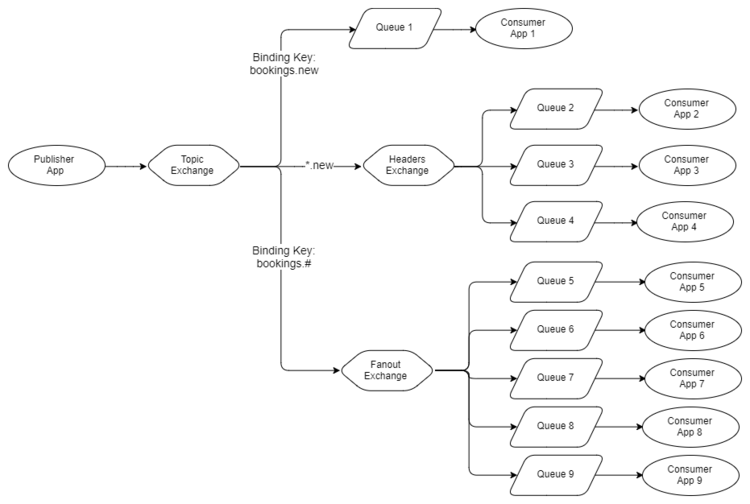
[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_005.png)

#### 2.3.2.1.Nhiều lớp Exchange

Để giảm chi phí định tuyến, phương pháp phân lớp có thể được áp dụng. Trong ví dụ dưới đây, chúng ta ban đầu định tuyến dựa trên một số lượng nhỏ các routing key hữu hạn đến các exchange khác. Mỗi ràng buộc cho một chủ đề làm tăng chi phí exchange đó. Trong trường hợp này, chúng ta có thể định tuyến tất cả các thông điệp đặt phòng đến một **Fanout Exchange**, nơi nó có thể được phát sóng hiệu quả đến tất cả consumer quan tâm.

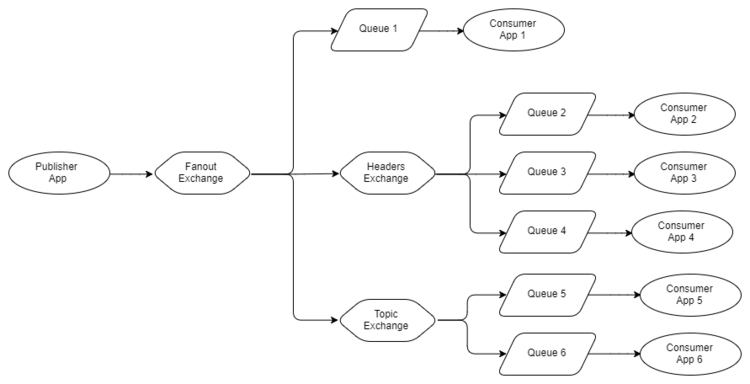
Tương tự, những consumer muốn lọc dựa trên các tiêu đề thư của một loại thông điệp nhất định có thể định tuyến thông điệp hiệu quả đến **Header Exchange** tốn kém hơn trong đó việc định tuyến dựa trên tiêu đề thư được thực hiện qua một tập hợp con các thông báo đi qua **Topic Exchange** .

*Hình 6:*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_006.png)

Nếu tất cả các exchange xuôi dòng muốn có thể định tuyến tất cả các thông điệp, điểm vào có thể là một Fanout thay thế.

*Hình 7:*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_007.png)

#### 2.3.2.1.Hệ thống định tuyến email với Header Exchange

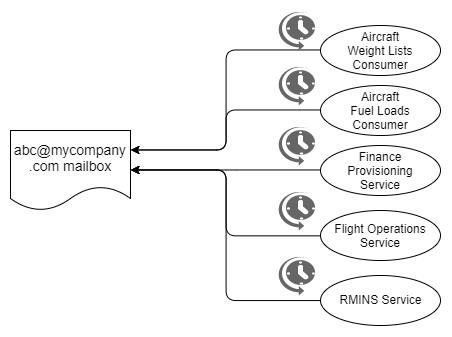
Định tuyến email không phải là một mẫu chung nhưng nó thể hiện sức mạnh của **Header Exchange**. Ví dụ này cũng cho thấy cách bạn có thể giảm sự phụ thuộc vào scheduler.

Giả sử chúng ta là một hãng hàng không và chúng ta làm việc với một đối tác tên là *ABC* thực hiện bảo dưỡng máy bay trên máy bay của chúng ta. Mỗi ngày hệ thống của chúng gửi cho chúng ta các email chứa thông tin bên trong email hoặc trong tệp đính kèm. Vâng chào mừng bạn đến với thế giới tích hợp thông qua SMTP, đó là sự thật.

Bạn có năm ứng dụng cần cập nhật hệ thống nội bộ với các tệp dữ liệu khác nhau mà ABC gửi cho chúng ta hàng ngày. Ví dụ, bộ phận tài chính cần biết tình trạng thành phần máy bay để tạo ra các mô hình dự đoán về chi phí trong tương lai để ngân sách cần thiết được cung cấp và hạch toán.

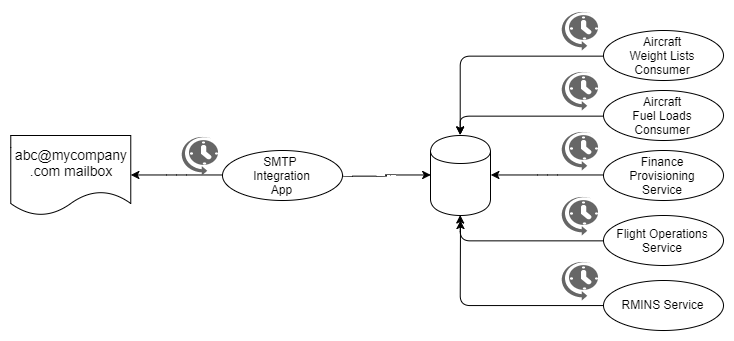
Khi tất cả năm ứng dụng đọc trực tiếp từ hộp thư, chúng ta không còn có thể dựa vào trạng thái đọc. Mỗi ứng dụng cần theo dõi những gì chúng đã đọc trước đó, bỏ qua các email mà chúng không quan tâm và được lập lịch để chạy mỗi X phút hoặc giờ. chúng ta phải thực hiện logic đọc hộp thư nhiều lần.

*Hình 8:*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_008.png)

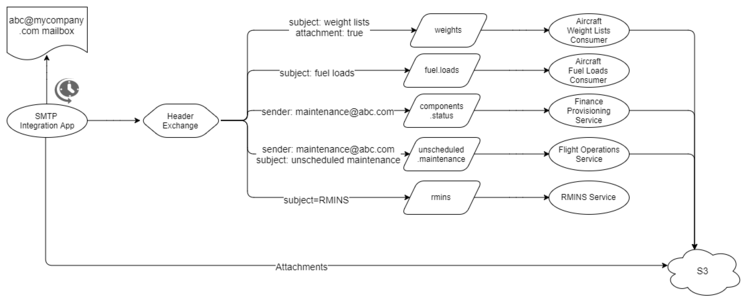
Thay vào đó, chúng ta có thể có một ứng dụng duy nhất chịu trách nhiệm đọc hộp thư ghi tất cả email và tệp đính kèm của chúng vào cơ sở dữ liệu. Sau đó, mỗi ứng dụng cần đọc từ cơ sở dữ liệu đó. Một lần nữa, mỗi ứng dụng phải theo dõi những email đã đọc, đó là đoạn code lặp đi lặp lại. Các ứng dụng này cũng cần được lên lập lịch bằng một scheduler

*Hình 9:*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_009.png)

Tùy chọn tốt hơn là có một ứng dụng được lên lịch duy nhất đọc từ hộp thư và gửi email dưới dạng thông điệp trên RabbitMQ, đến một **Header Exchange**. Tệp đính kèm có thể được lưu vào cơ sở dữ liệu hoặc dịch vụ đám mây như S3, chỉ với khóa đính kèm trong thông điệp. Các thuộc tính email như địa chỉ người gửi, địa chỉ người nhận, cc, chủ đề đều được thêm dưới dạng tiêu đề thư. Sau đó, mỗi ứng dụng chỉ cần tạo các ràng buộc phù hợp với các email mà chúng muốn tiêu thụ. Các ứng dụng không cần lịch trình khi chúng được đẩy các email từ RabbitMQ.

*Hình 10:*

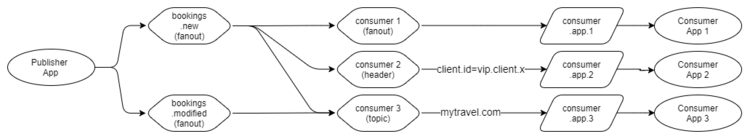
[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_010.png)

Hạn chế của **Header Exchange** là bạn chỉ có thể thực hiện các kết quả khớp chính xác. Điều này không loại trừ **Header Exchange** khá thường xuyên không may.

#### 2.3.2.1.Public Message Exchange, Private Consumer Exchange

Đây là một mô hình định tuyến dựa trên quy ước linh hoạt. Cấu trúc liên kết độc đáo có thể khó quản lý khi qui mô của chúng lớn hơn. Tôi có xu hướng thích các cấu trúc liên kết dựa trên quy ước vì chúng dễ quản lý. Trong pattern này, publisher của thông điệp khai báo **Fanout Exchange** dựa trên tên loại thông điệp. Consumer khi khởi động khai báo hàng đợi của riêng chúng và đối với mỗi thông điệp chúng tiêu thụ, chúng khai báo **private exchange** của chúng và liên kết nó với **message exchange** mà chúng muốn đăng ký. Sử dụng logic đơn giản này, các publisher và consumer tự động tạo ra tất cả các cơ sở hạ tầng hàng đợi mà không biết về nhau hoặc tác động lẫn nhau theo bất kỳ cách nào.

*Hình 11:*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_011.png)

Trong sơ đồ trên, publisher phát đi hai loại thông điệp: *new booking* và *modified booking*. Để định tuyến linh hoạt, nó thiết đặt kênh bán hàng là routing key (kênh bán hàng có thể là trang web chính, trang web so sánh, công ty du lịch, v.v.) và thêm một số dữ liệu thú vị khác trong tiêu đề của thông điệp.

Publisher đơn giản chỉ cần phát đi mỗi thông điệp đến exchange tương ứng của nó. Mỗi consumer đều có hàng đợi và exchange riêng. Nó liên kết exchange riêng của chính nó với messaging exchange này. Trong ví dụ của chúng ta, consumer app 1 muốn tất cả new booking. Consumer App 2 muốn tất cả new booking của một khách hàng cụ thể rất quan trọng. Consumer app 3 muốn tất cả new booking và modified booking liên quan đến *MyTravel.com* - nơi bán các đặt chỗ với tư cách là người bán bên thứ 3.

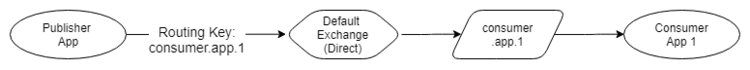
Pattern này tạo ra một cấu trúc liên kết tự quản lý trong đó yêu cầu làm sạch duy nhất là khi consumer bị xóa vĩnh viễn khỏi hệ thống. Triển khai và phát triển được đơn giản hóa khi tất cả các ứng dụng tạo ra các exchange, hàng đợi và liên kết RabbitMQ cần thiết mà chúng cần để giảm gánh nặng cho nhóm vận hành và các đường dẫn triển khai.

Một lợi ích khác của việc cung cấp cho mỗi consumer exchange riêng của chúng là các đội hỗ trợ có thể đặt *"theo dõi"* để xem tất cả các thông điệp được sử dụng bởi một ứng dụng. Bạn có thể tạo một hàng đợi và liên kết nó với exchange riêng của một ứng dụng consumer và nhận các bản sao của tất cả các thông điệp mà nó nhận được. Điều này cũng có thể được sử dụng để ghi log của một consumer nhất định

#### 2.3.2.1.Point-to-point Messaging

Chúng ta có thể bỏ qua các tùy chọn định tuyến khác nhau và gửi thông điệp trực tiếp đến hàng đợi theo tên. Gửi thông điệp đến **Default Exchange**, với tên của hàng đợi là routing key và nó sẽ được chuyển thẳng đến hàng đợi.

*Hình 12:*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_012.png)

Điều này hữu ích khi publisher muốn kiểm soát consumer nào xử lý thông điệp, thay vì dựa vào định tuyến trong đó 0 đến nhiều hàng đợi có thể nhận được thông điệp. **NServiceBus** sử dụng **Default Exchange** để gửi command. **NServiceBus** chia các thông điệp thành hai loại: event và command. Event được khai báo để trao đổi nơi bất kỳ consumer nào cũng có thể đăng ký tham gia sự kiện. Các ứng dụng gửi command trực tiếp đến consumer cụ thể bằng cách sử dụng tên hàng đợi của chúng

#### 2.3.2.1.Định tuyến phân cấp

Đây là một phần mở rộng của mẫu **Public Message Exchange**, **Private Consumer Exchange** và cho phép **Topic Exchange** như định tuyến bằng cách sử dụng **Fanout Exchange**.

Hãy tưởng tượng chúng ta đã chia nghiệp vụ của chúng ta thành các domain, sub-domain và action. Chúng ta có thể xây dựng một không gian tên thông điệp theo định dạng domain.sub-domain.action:

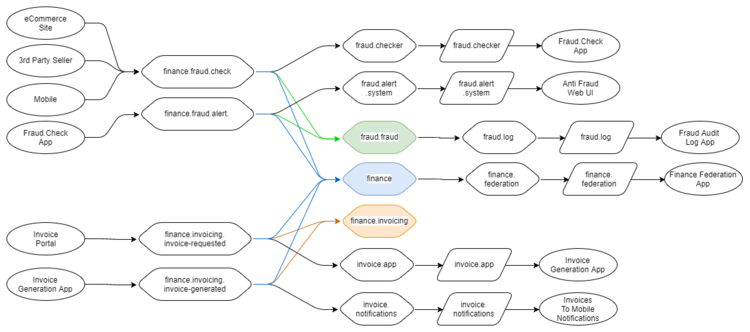
* finance.invoicing.invoice-requested
* finance.invouring.invoice-created
* finance.fraud.alert
* finance.fraud.check

Chúng ta tạo thêm message exchange:

* finance.invoicing
* finance.fraud
* finance

Chúng ta tạo các liên kết cho các exchange này theo không gian tên như dưới đây:

Hình 15:

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_015.png)

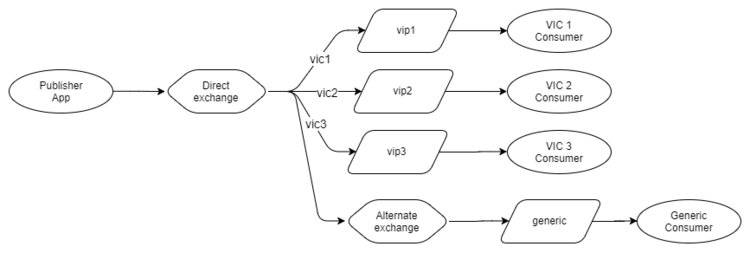
Pattern này có thể hữu ích khi bạn muốn nắm bắt thông điệp của các nhóm lớn các exchange liên quan mà không phải tạo ra số lượng lớn các liên kết. Khi các publisher khai báo exchange của thông điệp mà chúng publish, thì chúng cũng khai báo các exchange trong hệ thống phân cấp và các liên kết cần thiết. Điều này có nghĩa là một khi bạn đăng ký exchange cha, khi các exchange con mới được thêm vào, thông điệp của chúng sẽ tự động được chuyển đến bạn.

#### 2.3.2.1.Định tuyến If/Else với Alternate Exchange

Hãy tưởng tượng chúng ta có một số khách hàng siêu quan trọng yêu cầu hành vi tùy chỉnh cho từng thông điệp và mỗi thông điệp này cần đến một consumer dành riêng cho khách hàng đó. Các thông điệp liên quan đến hàng trăm khách hàng ít quan trọng hơn sẽ được xử lý bởi một consumer chung chung.

Chúng ta có thể đạt được điều này bằng cách đặt một mã định danh khách làm routing key và gửi thông điệp đến một **Direct Exchange** được cấu hình với một **Alternate Exchange**.

*Hình 18:*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_018.png)

**Alternate Exchange** chỉ là một exchange thông thường trong bốn loại exchange. Bạn thậm chí có thể xâu chuỗi các exchange luân phiên với nhau và thực hiện logic if, else if, else.

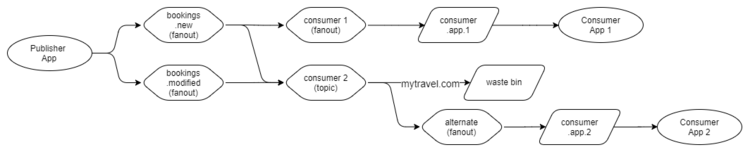
**Topic Exchange** không thể cung cấp định tuyến này vì nó không thể thực hiện OR, nó chỉ có thể thực hiện AND. Nếu chúng ta sử dụng **Topic Exchange** và sử dụng ký tự # đại diện để chụp tất cả các thông điệp, cuối cùng chúng ta sẽ xử lý các thư khách hàng rất quan trọng cũng như các máy khách ít quan trọng hơn.

#### 2.3.2.1.KHÔNG ĐỊNH TUYẾN với Alternate Exchange

tiêu cực. Thay vào đó, chúng ta có thể tạo một hàng đợi *"thùng rác"* và liên kết cho thông điệp bạn không muốn. Hàng đợi này được thiết lập với một thông điệp dựa trên hàng đợi rất ngắn TTL để các thông điệp bị loại bỏ gần như ngay lập tức khi đến nơi.

Bạn định cấu hình exchange với một **Alternate Exchange** và kết nối khách hàng của bạn với một hàng đợi liên kết với **Alternate Exchange** đó. Bây giờ bạn tiêu thụ tất cả các thông điệp ngoại trừ một loại bạn không muốn.

*Hình 19:*

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_019.png)

Cấu trúc liên kết này tương tự như cấu trúc trong mẫu **Public Message Exchange**, **Private Consumer Exchange**. Consumer 2 thiết lập một **Topic Exchange** private và định tuyến tất cả các *booking* được thực hiện từ kênh bán hàng *mytravel.com* đến hàng đợi *"rác"* và sau đó tiêu thụ phần thông điệp còn lại.

Rõ ràng bạn chỉ có thể để consumer của mình tiêu thụ mọi thông điệp và chỉ cần loại bỏ những thông điệp bạn không muốn. Nó phụ thuộc vào kịch bản cụ thể của bạn.

#### 2.3.2.1.Delayed Retry với Ephemeral Exchanges và Queue

Chúng ta có thể đạt được kết quả tương tự như pattern trước đó với các **Ephemeral Exchange** và hàng đợi không bền. Khi một ứng dụng muốn gửi thông điệp cho Delayed retry, nó sẽ tạo ra một exchange và queue một lần với tên được đảm bảo duy nhất (ví dụ sử dụng GUID/UUID).

**Ephemeral Exchange** được cấu hình như sau:

* Fanout
* Auto-delete

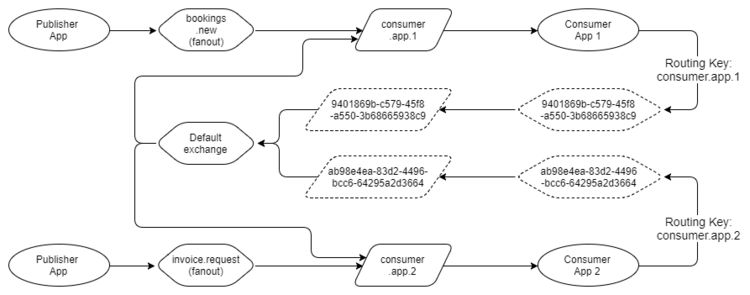
Hàng đợi không bền được cấu hình như sau:

* Một thông điệp TTL tương ứng với độ trễ bạn muốn.
* DLX của nó là **Default Exchange**.
* Một hàng đợi hết hạn một vài giây sau thông điệp TTL.
* Một liên kết đến **Ephemeral Exchange**.

Consumer khai báo exchange và hàng đợi, sau đó gửi thông điệp cho Delayed retry với routing key là tên của hàng đợi của chính nó. Một loạt các sự kiện tiếp theo xảy ra:

1. **Ephemeral Exchange** định tuyến thông điệp đến hàng đợi không bền
2. Thông điệp nằm trong hàng đợi cho khoảng thời gian thông điệp TTL
3. Thông điệp bị chết được gửi đến **Default Exchange**
4. **Default Exchange** định tuyến thông điệp này đến hàng đợi khớp với routing key
5. Hàng đợi không bền vững đạt đến TTL của hàng đợi và tự động xóa chính nó
6. **Ephemeral Exchange** thấy rằng không có hàng đợi nào liên kết với nó và nó tự động xóa chính nó.

Hình 21:

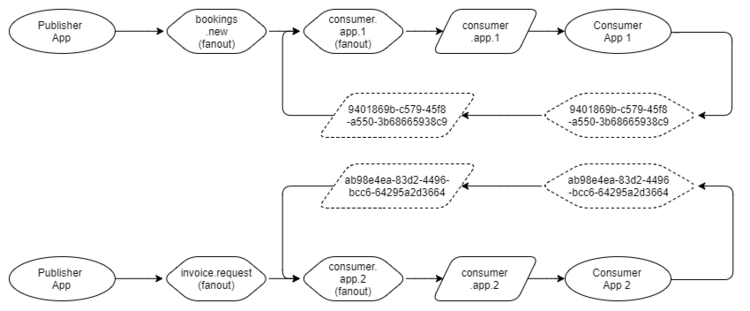
[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_021.png)

Cân nhắc cần tính đến:

* Tạo exchange, hàng đợi và liên kết là tương đối tốn kém. Nếu bạn tạo tải cao khi thử lại thì điều này có thể gây áp lực quá lớn cho cluster.
* Giống như pattern exchange xếp tầng, routing key ban đầu và thông điệp TTL được loại bỏ để thực hiện công việc này.

Nếu bạn sử dụng pattern **Public Message Exchange**, **Private Consumer Exchange pattern**, bạn hoàn toàn không cần dựa vào **Default Exchange** và có thể định cấu hình exchange riêng của consumer làm DLX. Điều này loại bỏ sự cần thiết phải thiết lập một routing key tùy biến.

Hình 22:

[](https://gist.githubusercontent.com/dodangquan/187202483a2cfe96fd4a549039929e84/raw/f59d3474ca812276b28b3adf8ce7a68ea18d7630/z_img_022.png)

# **CHƯƠNG 3: Xây dựng demo sử dụng phương thức AMQP trong RabbitMQ với AspCore 3.1**

**3.1 ASP.NET Core là gì**

ASP.NET Core là một open-source mới và framework đa nền tảng (cross-platform) cho việc xây dựng những ứng dụng hiện tại dựa trên kết nối đám mây, giống như web apps, IoT và backend cho mobile.

Ứng dụng ASP.NET Core có thể chạy trên .NET Core hoặc trên phiên bản đầy đủ của .NET Framework. Nó được thiết kế để cung cấp và tối ưu development framework cho những dụng cái mà được triển khai trên đám mây (clound) hoặc chạy on-promise.

Nó bao gồm các thành phần theo hướng module nhằm tối thiểu tài nguyên và chi phí phát triển, như vậy bạn giữ lại được sự mềm giẻo trong việc xây dựng giải pháp của bạn. Bạn có thể phát triển và chạy những ứng dụng ASP.NET Core đa nền tảng trên Windows, Mac và Linux.

**Khác biệt quan trọng của ASP.NET và ASP.NET Core**

|  |  |
| --- | --- |
| **ASP.NET** | **ASP.NET CORE** |
| Phiên bản hiện tại 4.8 | Phiên bản hiện tại 3.1 |
| Nền tảng đã có từ lâu | Hoàn toàn được thiết kế mới |
| Chạy trên .NET Framwork | Chạy trên cả .NET Core và .NET Framework |
| Chỉ trên Windows | Chạy trên tất cả các OS sử dụng .NET Core |
| Nền tảng ổn định với tính năng phong phú | Chưa hoàn chỉnh nhưng mong đợi sẽ hoàn chỉnh trong tương lai |
| WebForms được hỗ trợ | Không hỗ trợ WebForms |
| System.web.dll cồng kềnh | Nhỏ, nhẹ và module hóa |
| Bản quyền của Microsoft | ASP.NET Core là mã nguồn mở |

**3.2 Xây dựng demo**

### **3.2.1 Đặt Vấn Đề**

Trong một giao dịch chuyển tiền trong một hệ thống microservice ta cần lưu lại lịch sử giao dịch đó (3 trạng thái : đang thực thi, thành công , thất bại ; 2 cách thức chuyển tiền nhanh và chuyển tiền chậm) .Ta sẽ có 3 service : service 1 sẽ tiếp nhận giao dịch và service 2 sẽ lưu lại lịch sử của phiên giao dịch này vào database ,service 3 sẽ lấy data trên service 2 để thực hiện logic thanh toán chuyển các giao dịch đang thực thi với cách thức là chuyển tiền chậm . Service 1 ,Service 2 và Service 3 sẽ giao tiếp với nhau trên giao thức AMQP sử dụng RabbitMQ .

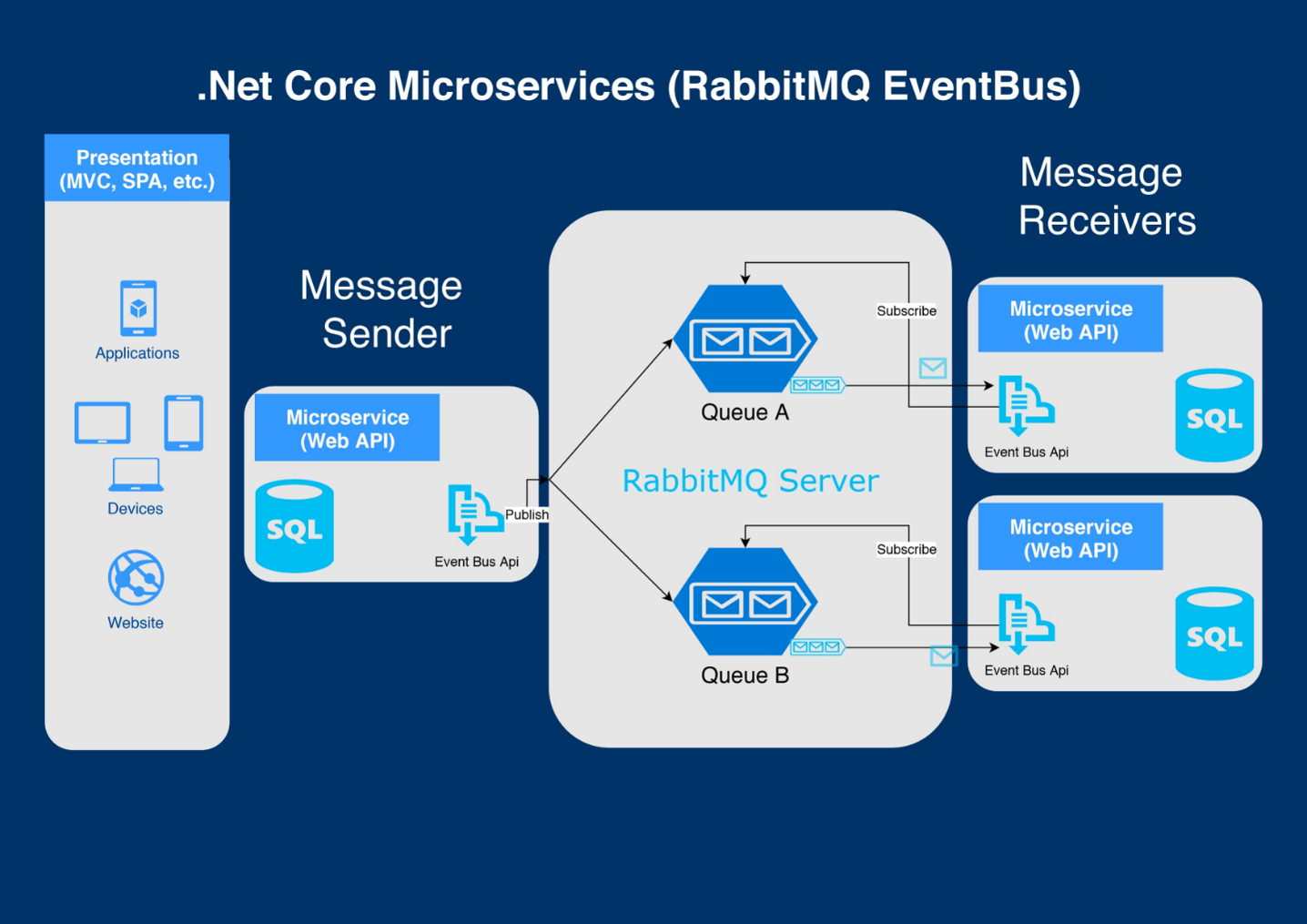
### **3.2.1 Thiết kế**

Service 1 : Database TranferDb - Table Tranfer

|  |  |  |
| --- | --- | --- |
| **COLUMN NAME** | **DATA TYPE** | **ALLOW NULLS** |
| ID | INT | NOT NULL |
| FROM ACCOUNT | NVARCHAR(MAX) | NOT NULL |
| TO ACCOUNT | NVARCHAR(MAX) | NOT NULL |
| TRANSFER AMOUNT | DECIMAL(18, 2) | NOT NULL |
| PAYMENT TYPE | INT | NOT NULL |
| PAYMENT STATUS | INT | NOT NULL |

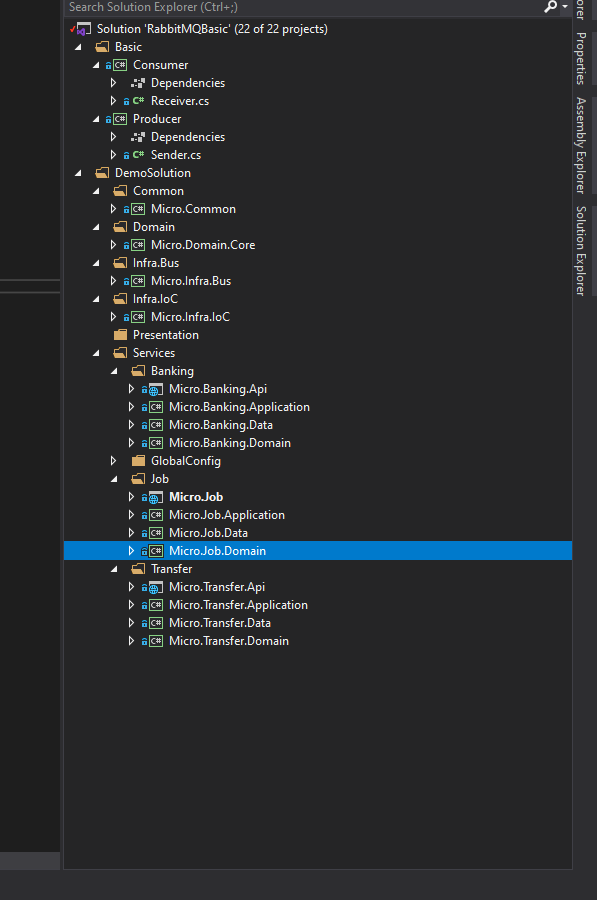
Service 2 : Database Bankingdb - Table account

|  |  |  |
| --- | --- | --- |
| **COLUMN NAME** | **COLUMN NAME** | **COLUMN NAME** |
| ID | INT | NOT NULL |
| ACCOUNT TYPE | NVARCHAR(MAX) | NOT NULL |
| ACCOUNT BALANCE | DECIMAL(18, 2) | NULL |

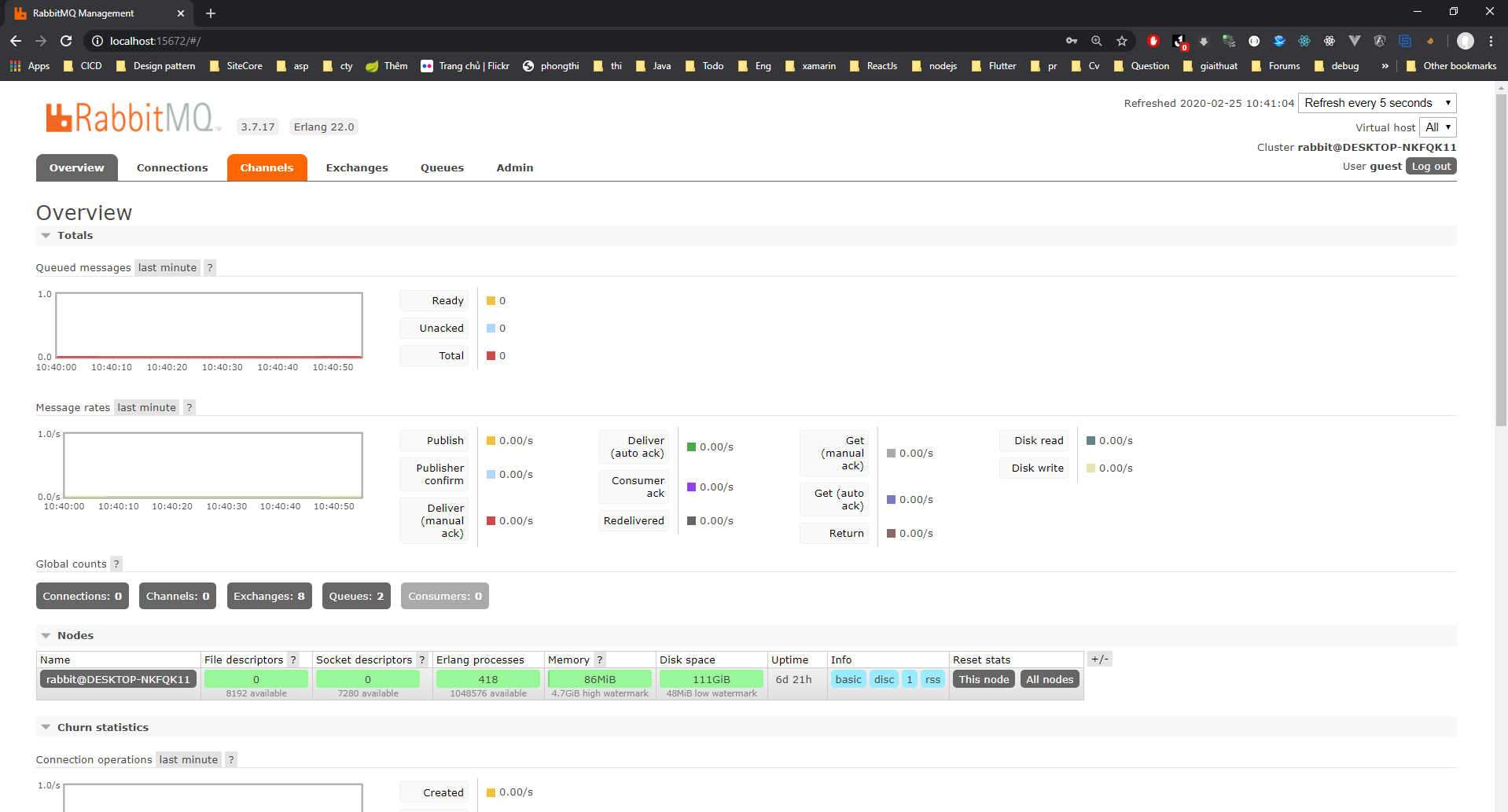


Hình dsfd

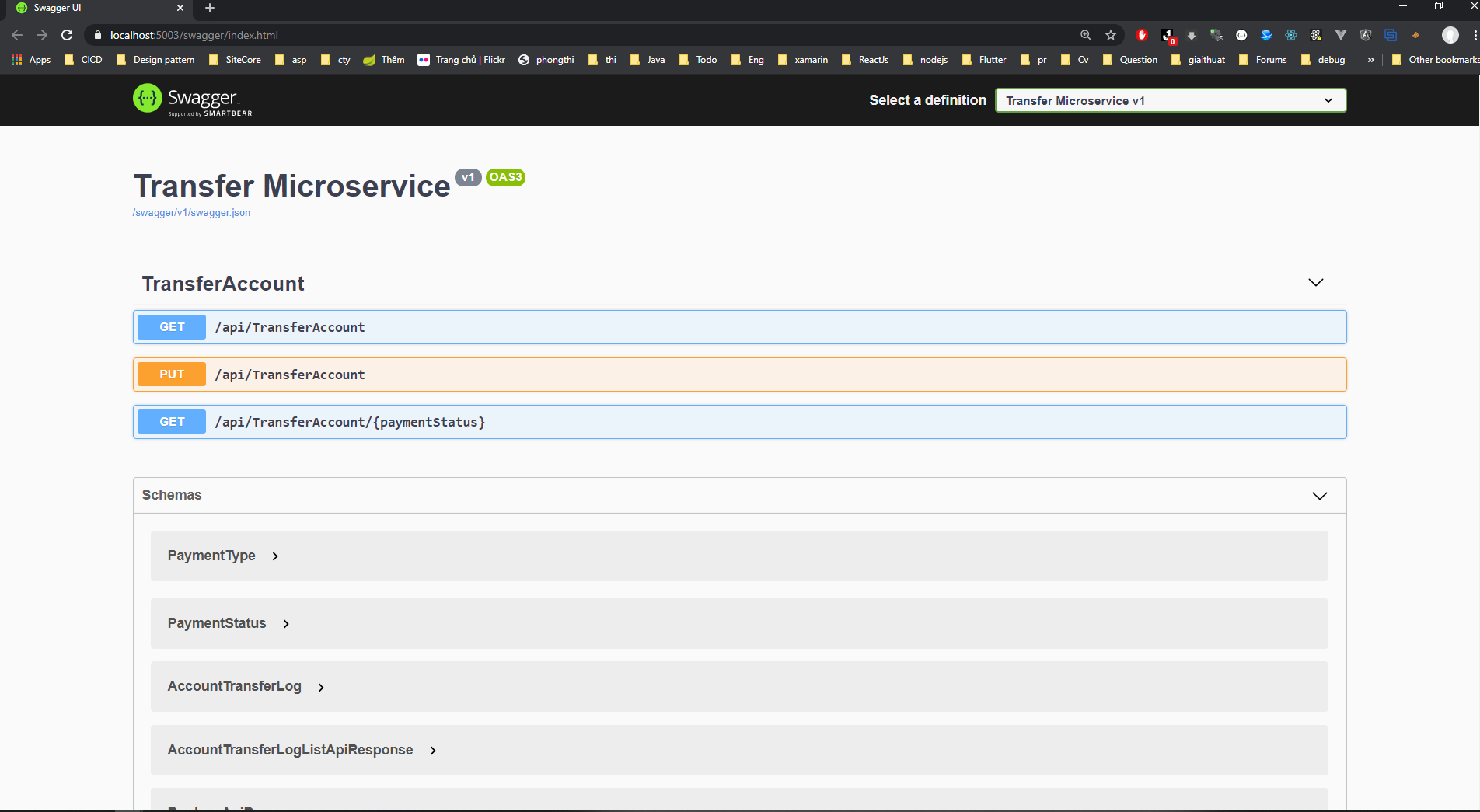
### **3.2.1 Giao Diện – Mô Tả**



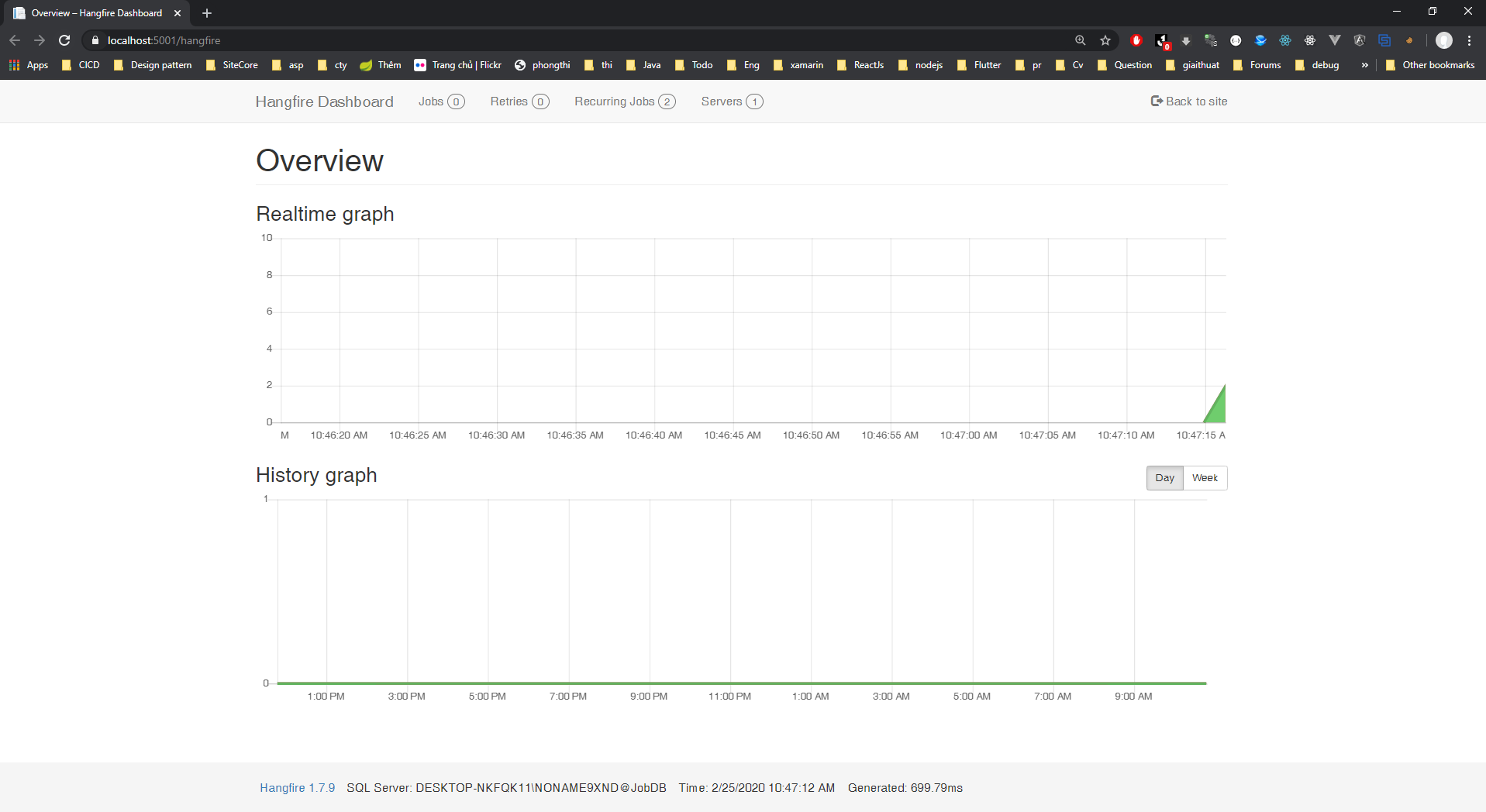
Hình dfgfsdg

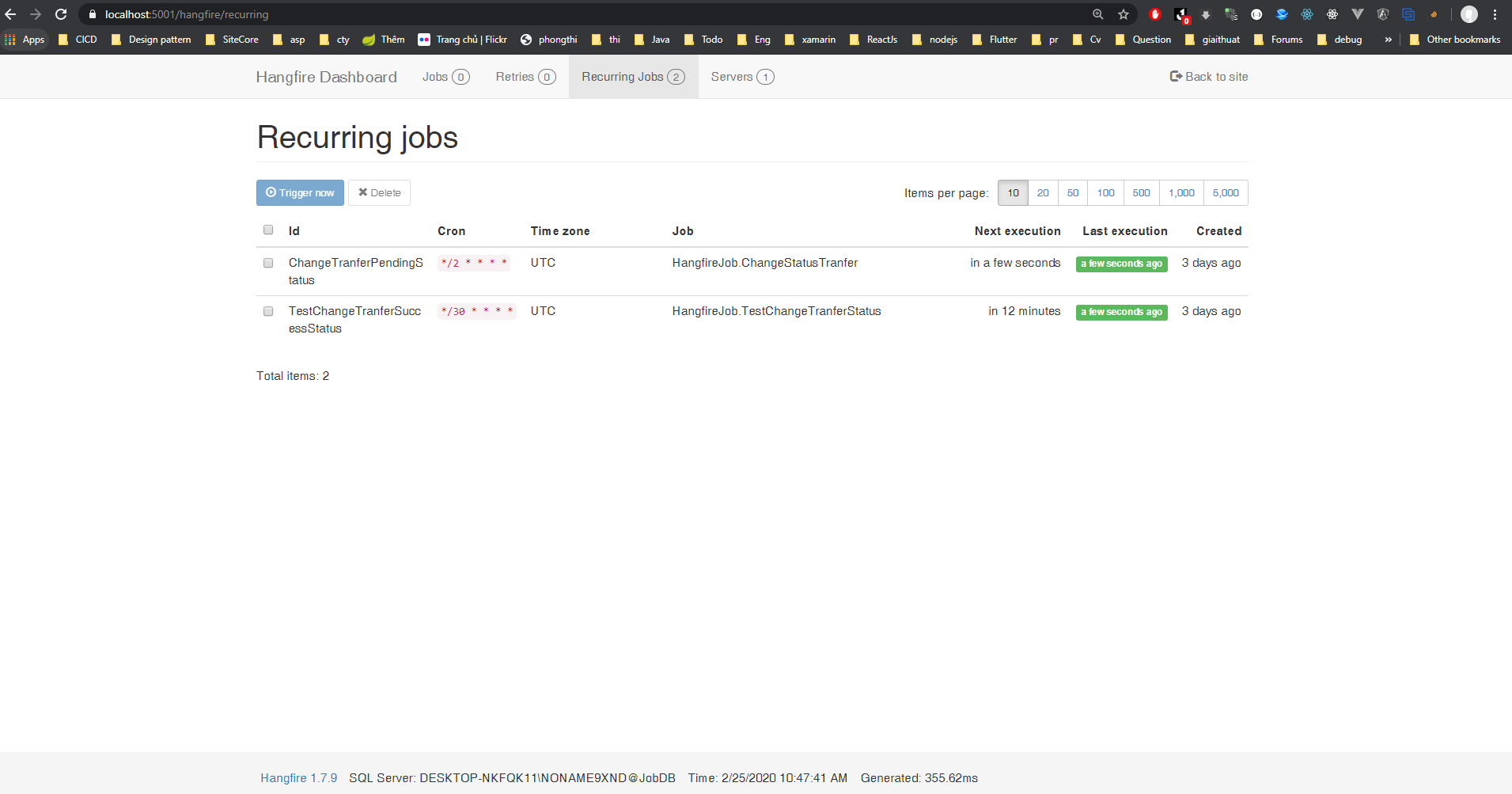


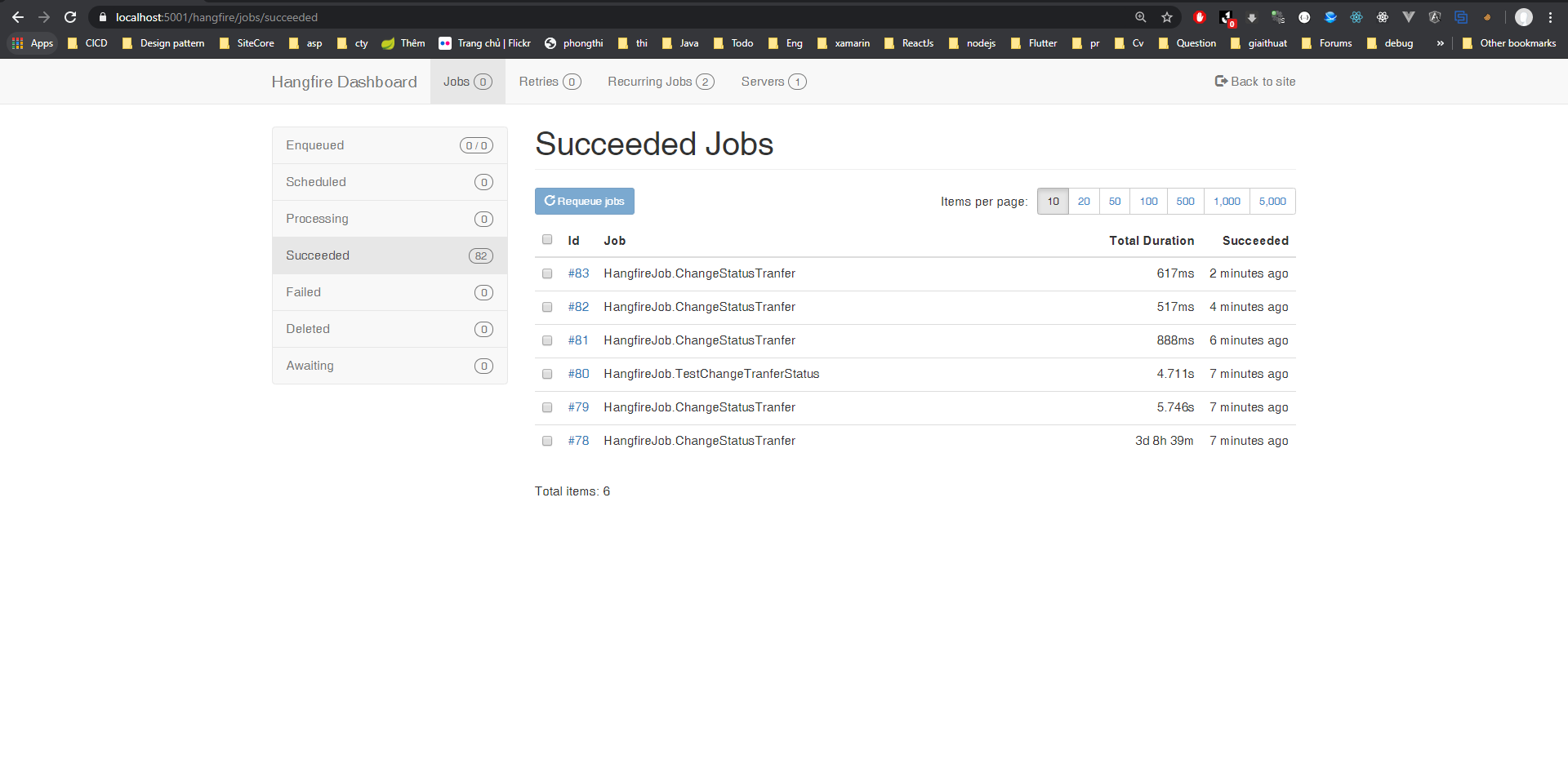
Hình ,,



Hình sdf







# **TÀI LIỆU THAM KHẢO**

<https://tapit.vn/5-giao-thuc-truyen-tai-du-lieu-trong-internet-things/>

<https://medium.com/@buihuycuong/nh%E1%BB%AFng-%C4%91i%E1%BB%81u-c%E1%BA%A7n-bi%E1%BA%BFt-v%E1%BB%81-rabbitmq-354a37ecf218>

<https://kipalog.com/posts/Tim-hieu-RabbitMQ---Phan-1>  
<https://seal.deha.vn/cac-giao-thuc-tich-hop-he-thong-pho-bien/>

<https://viblo.asia/p/tim-hieu-ve-rabbitmq-OeVKB8bMlkW>

<http://alexvolov.com/2016/06/amqp/>

<https://smartfactoryvn.com/resources/technical-writer/rabbitmq-la-gi/>

<https://www.dds-foundation.org/>

<https://www.dds-foundation.org/why-choose-dds/>

<https://dodangquan.blogspot.com/2018/11/rabbitmq-va-kafka-phan-2-cau-truc-cua-rabbitmq-va-cac-messaging-pattern-khi-su-dung-rabbitmq.html>